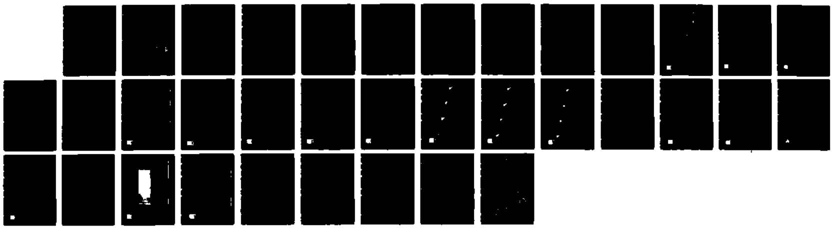
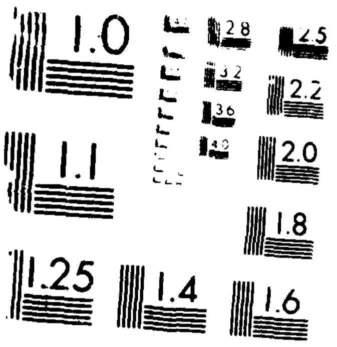
AD-A180 574 PARALLELISM IN THE EXECUTION OF A ROUTINE KNOWLEDGE 1/1
RULE SYSTEM ON THE BUTTERFLY (TRADEMARK) COMPUTER(U)
BBN LABS INC CAMBRIDGE MA A BOULANGER DEC 86 BBN-6436
UNCLASSIFIED MDA903-84-C-0033 F/G 12/5 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

②

BBN Laboratories Incorporated

A Subsidiary of Bolt Beranek and Newman Inc.

Report No. 6436

Parallelism in the Execution of a Routine Knowledge Rule System on the Butterfly™ Computer

December 1986

Prepared for:
Defense Advanced Research Projects Agency

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER Report No. 6436	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Parallelism in the Execution of a Routine Knowledge Rule System on the Butterfly Computer		5. TYPE OF REPORT & PERIOD COVERED Technical Report December 1986
		6. PERFORMING ORG. REPORT NUMBER 6436
7. AUTHOR(s) Albert Boulanger		8. CONTRACT OR GRANT NUMBER(s) MDA093-84-C-0033
9. PERFORMING ORGANIZATION NAME AND ADDRESS BBN Laboratories Inc. 10 Moulton Street Cambridge, MA 02238		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE December 1986
		13. NUMBER OF PAGES 29
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Knowlwdge Rule Systems, Parallel Computing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This project ported a routine knowledge rule system to the Butterfly(TM) mutiprocessor. The goal was to explore parallelization techniques with an existing rule system originally written for serial execution. The rule system was rewritten to introduce parallelism, and run on a single processor to establish a benchmark for serial operation. The same version, with parallelism enabled, was then run on a 16-node Butterfly multiprocessor. The metering tools on the Butterfly were used to display task behavior and

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

processor utilization. The information gained from these displays was used to guide further experimentation with the granularity of the rules and with the system code to investigate bottlenecks that were lengthening execution time.

The project demonstrated that parallelization of routine knowledge rule systems can yield substantial speedup. It also demonstrated that the metering tools on the Butterfly can be used to achieve additional speedup of parallel implementations. The implications of this research are discussed and compared to the findings of research at Carnegie-Mellon University on parallelizing production systems.

Report No. 6436

PARALLELISM IN THE EXECUTION OF A ROUTINE KNOWLEDGE RULE SYSTEM ON THE BUTTERFLY^{TM} COMPUTER

Albert Boulanger

December 1986

Prepared by:

BBN Laboratories Incorporated
10 Moulton Street
Cambridge, Massachusetts 02238

Prepared for:

Defense Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

Table of Contents

1. Abstract

This project ported a routine knowledge rule system to the Butterfly™* multiprocessor. The goal was to explore parallelization techniques with an existing rule system originally written for serial execution. The rule system was rewritten to introduce parallelism, and run on a single processor to establish a benchmark for serial operation. The same version, with parallelism enabled, was then run on a 16-node Butterfly multiprocessor. The metering tools on the Butterfly were used to display task behavior and processor utilization. The information gained from these displays was used to guide further experimentation with the granularity of the rules and with the system code to investigate bottlenecks that were lengthening execution time.

The project demonstrated that parallelization of routine knowledge rule systems can yield substantial speedup. It also demonstrated that the metering tools on the Butterfly can be used to achieve additional speedup of parallel implementations. The implications of this research are discussed and compared to the findings of research at Carnegie-Mellon University on parallelizing production systems.

2. Introduction

As LISP implementations on parallel machines become available to the artificial intelligence community, work is beginning on exploring the introduction of parallelism to existing algorithms and the development of new algorithms to exploit the processing power of parallel machines. The work reported on here is part of a larger DARPA-funded parallel AI tools project[1] that is designed to accelerate that process by building tools that will facilitate artificial intelligence programming on parallel multiprocessor machines.

This effort involves experimentation with the new parallel programming primitives to develop appropriate techniques for incorporating them into higher level functions. A side effect of this work will be to determine whether the available primitives are sufficient and what additional primitives might be required to fully exploit parallelism in various application areas.

This project investigated the use of parallelism with a set of rule packets typical of a routine knowledge rule system. A routine knowledge rule system is one that captures routine expertise. Rules in a routine knowledge rule system are represented as situation-action pairs. Related rules are organized into rule packets. The rule system consisted of 221 rules in 51 rule packets. The rules in this rule system were particularly suitable for parallelization because they were totally independent of each other. In other applications it would be likely that there would be some serial dependencies between rule packets and this would have to be taken into account in the parallelization process.

* Butterfly is a trademark of BBN Laboratories Inc.

The rules used were classification rules. The data for a given execution of the rule system is 51 items of information, on the basis of which the classification decision is made. Each rule packet corresponds to one of these items. Characteristic of this application is that very few of the rules actually fire for any single data set. The data set (Data Set A) used during most of this project fired only six rules of the 221 rules in the system. The other data set (Data Set B) used in the project fired 15.

The program uses rules that were developed under the rule-based system FLEX[2]. Rules in FLEX are organized into rule packets. Each packet is a description which consists of a set of rules, local variables, and a list of arguments. The rules within a rule packet have access to all of the packet's local variables and arguments. Instances of rule packets, once they are stable, can be expanded and compiled to run as functions.

The rule packets in this application are all FLEX DO-ALL type rule packets. DO-ALL type rule packets run each rule in the packet, regardless of how many rules fire. The first value returned by a rule packet is NIL if no rule fired and non-NIL if any rule fired. The other values returned by a DO-ALL rule packet are the values of the local variables in the packet. In this application, each rule packet comprises the rules applied to a given item in the data set. The work of classifying the subject of the data set is accomplished by side effect.

The FLEX system does implement uncertainty, but uncertainty is only used in a limited way in these rule packets. It is used to deal with situations in which there is a missing item in the data set.

The rule system was originally written in Zetalisp, using the Flavors facility. For the AI tools project, its kernel was rewritten in Common Lisp, using the Common Loops facility. For this research, the kernel of the rule system was again rewritten so that the rules would expand into SCHEME for execution on the Butterfly. A simple version of the database was also implemented in SCHEME. In the SCHEME version, a rule packet compiles into a function. So, in effect, the 51 rule packets in this system compiled into 51 SCHEME functions which could run independently of each other. Parallelism was also introduced within rule packets.

3. Constraints

There were two constraints on this project, one self-imposed and one a hardware limitation. The hardware limitation was that the available Butterfly multiprocessor was a 16-node machine. This set a limit on the concurrency that could be achieved. The self-imposed constraint was that we elected not to attempt to achieve parallelism within individual rules in order to narrow the scope of the project.

It should also be noted that our goal was not to achieve maximum speedup. It was investigating bottlenecks that would be a barrier to maximum speedup that interested us.

4. Metering Tools

The figures used in this report are screen images produced from the display of a Symbolics 3600 used as a front-end machine on the Butterfly. The display was produced by the Butterfly LISP metering tools that have been developed to display a range of information about the execution of a program on the system.

Our experience with this project demonstrated the value of good metering tools on a parallel multiprocessor and the importance of effective display of the information they provide. The task and performance data made available by the metering displays available in the Butterfly LISP environment not only demonstrated performance improvements, but suggested areas in which additional improvements could be made.

A good example of the use of the metering tools can be seen in Figure 3. The top of the display (to the right of the BBN logo) is the metering pane of the user interface window. It consists of a horizontal strip divided into a number of rectangles. The top section of each rectangle shows the current state of the processor. The bottom section of each rectangle represents how full the heap is on that processor. In these figures, the metering pane is meaningless because it displays real time information during the actual display of the other panes. The metering tools that created the rest of the display recorded data during the rule system execution, saved it, and then displayed it upon request.

The center of the display is the task graph pane of the task display window. Each horizontal bar represents a task. When the bar has a light, hatched pattern, the task is queued and will run as soon as a processor is available. When the bar has a dark, crosshatched pattern, the task is running on a processor. A light, dotted pattern indicates the task is suspended and waiting for the results of another task. The task that turned on the metering and any tasks spawned previous to that task are not metered and therefore not displayed.

Spawning and data dependency are indicated by arrows from one task to another. An arrow drawn from task A to the beginning of task B means that task A spawned task B. An arrow drawn from task A to task B at the transition of task A from running (dark, crosshatched area) to waiting (light, dotted area) means that task A is suspended while waiting for task B to complete. An arrow drawn from the end of task B to task A at the transition of task A from waiting (light, dotted area) to queuing (light, hatched area) means that task A was waiting for task B to complete and is now on the queue and will run as soon as a processor is available. An arrow drawn from the end of task B to task A at the transition of task A from waiting (light, dotted area) to running (dark, crosshatched area) means that task A was waiting for task B to complete and has now resumed running.

The user can zoom to display details and scroll through the task graph pane. In Figure 3, the user has zoomed in on a section of the task graph pane in order to see in more detail the activity and transitions of a group of tasks. In most of the figures used in this report, the display has been scrolled to show the rule system tasks. .

The bottom of the display is the task summary pane of the task display window. It can be used to display a variety of information. A pop-up menu is available to select the appropriate option. The user can choose to show graphs of active tasks, queued tasks, waiting tasks, or total tasks. The user can also edit various features of the curves used to display the task information. In Figure 3, the user has chosen to view a summary of waiting tasks and active tasks, using a dashed-line curve.

The panes of the display are mouse-sensitive and can be manipulated to provide a variety of graphic and text displays of information. Further details on the metering tools displays are available with the Butterfly LISP documentation.

5. Introducing Parallelism

The next step was to rewrite the rule system to introduce parallelism. To accomplish this, the FUTURE special form was wrapped around each rule packet. FUTURE is a concept found in the Multilisp language developed at MIT. It has been implemented in Butterfly LISP and is described in the Butterfly LISP Reference Manual[3]. FUTURE creates a data structure as a placeholder for the value of the expression it encloses and then places a task to compute this value on the system-wide task queue.

6. Serial Benchmark

We then had to establish a benchmark for serial execution of the rule system. We rewrote the rule system first and then *turned off* the parallelism so that the initial parallel runtime could be meaningfully compared to the serial runtime. Figure 1 displays the run time of four different executions of the rule system on a single processor with the same data. We will refer to this data throughout the rest of this report as Data Set A. Actual processing is indicated by the dark crosshatched area. It is displayed against time, which reads left to right.

The last serial execution took much longer than the first three runs. This appears to be the result of an unfavorable distribution of data in memory after garbage collection. We chose to ignore this last run, so that any speedup in the parallel version would not be exaggerated. The average runtime for the first three serial execution was 50.5 seconds.

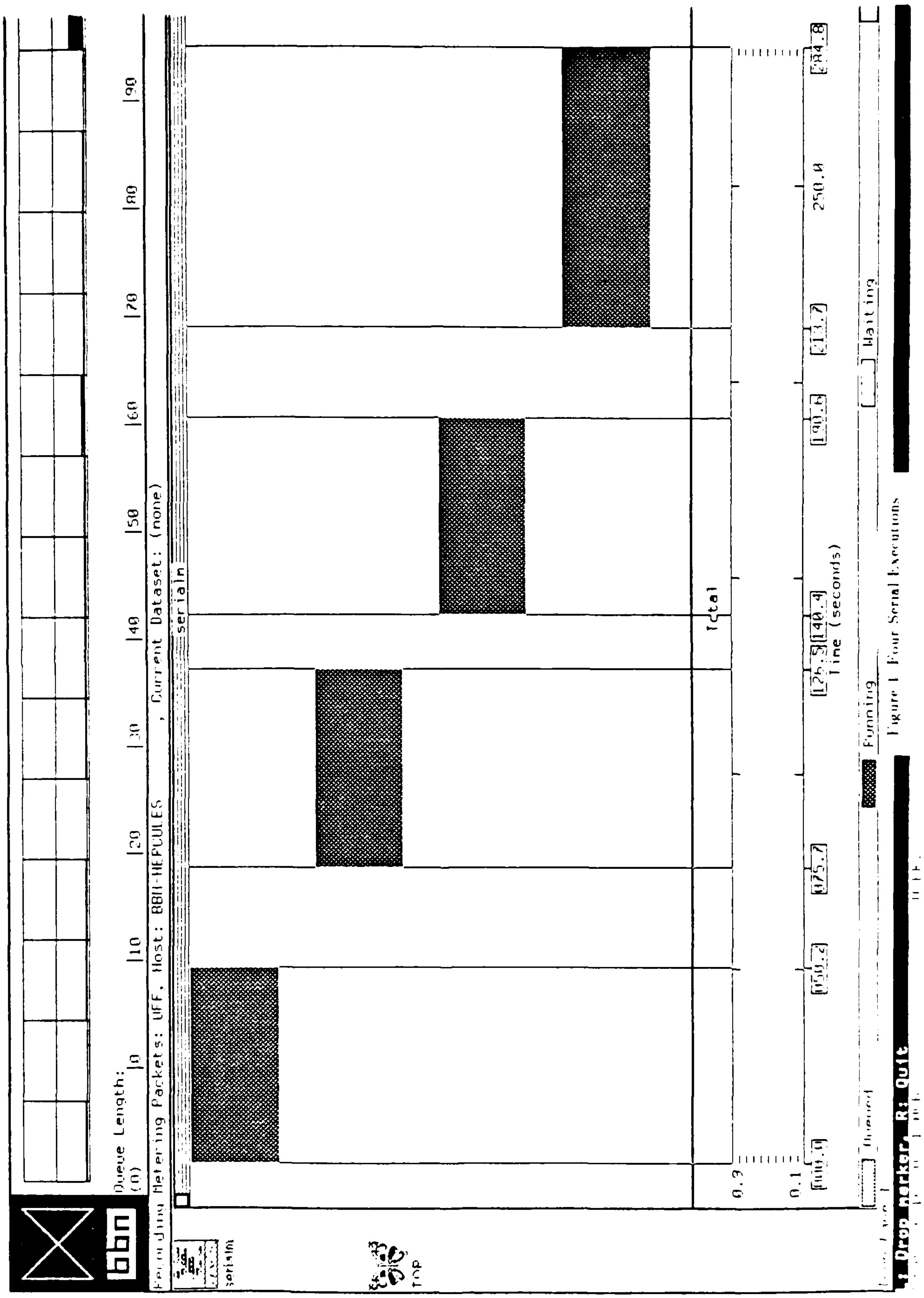


Figure 1 Four Serial Executions

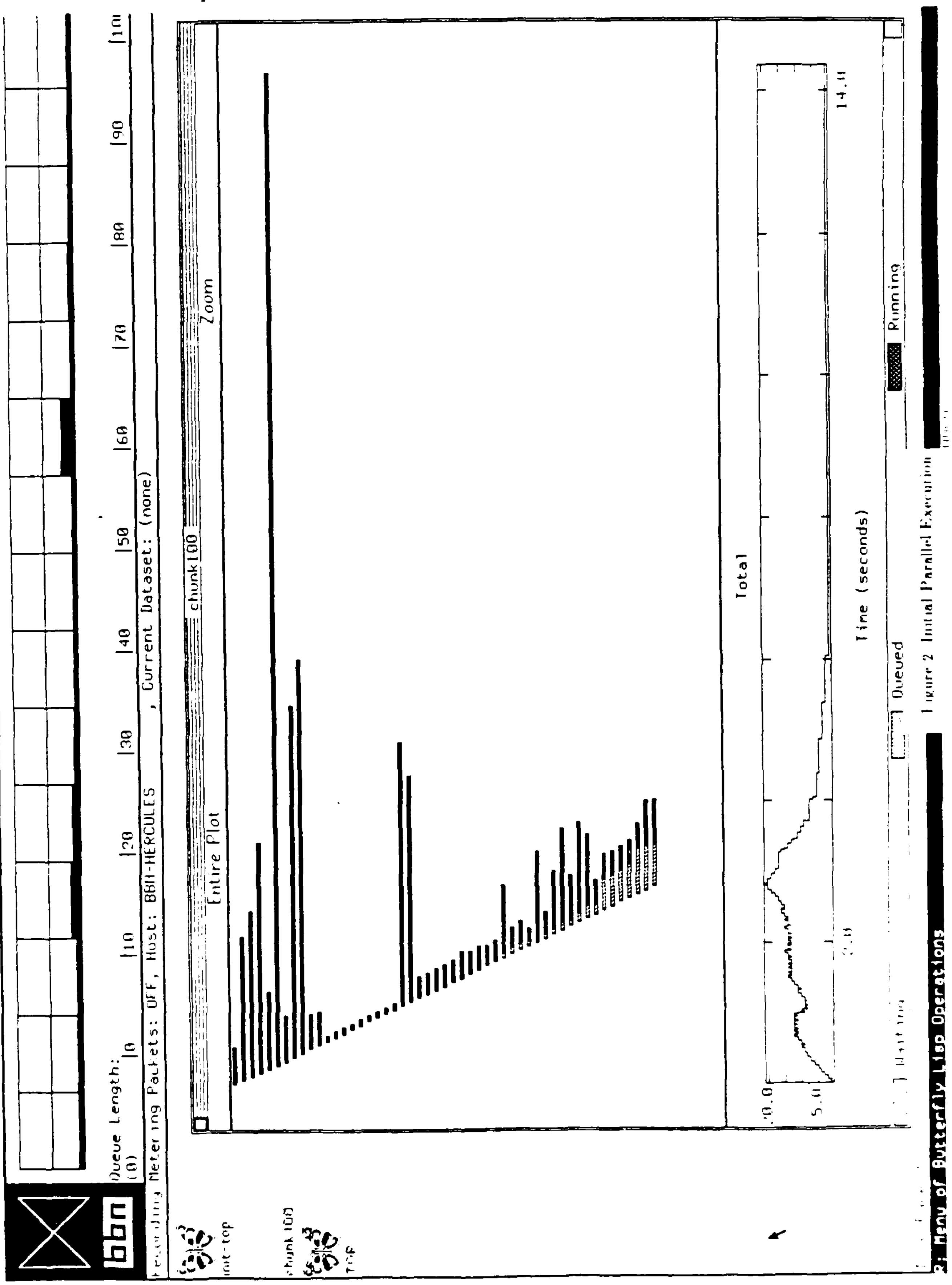
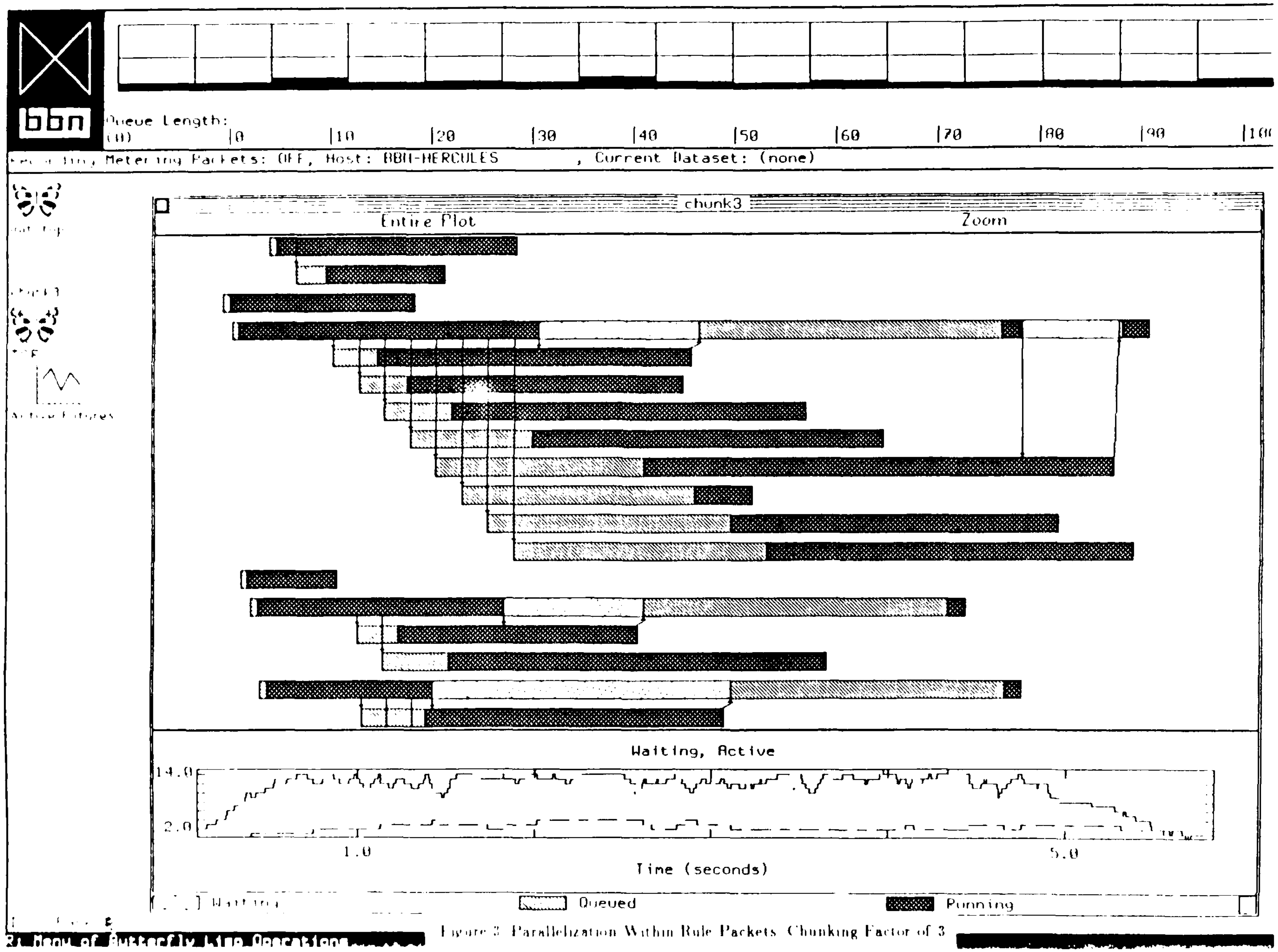


Figure 2 Initial Parallel Execution


Queue Length: (0) |0 |10 |20 |30 |40 |50 |60 |70 |80 |90 |100

Metering Packets: OFF, Host: BBN-HERCULES , Current Dataset: (none)

Figure 3: Parallelization Within Rule Packets: Chunking Factor of 3

7. Parallel Benchmark

Figure 2 shows the first run of each of the rule packets in parallel. Each horizontal bar on the graph represents one of the 51 rule packets. The run time for the system in parallel was just over 14 seconds. This represents a performance improvement over the serial version by a factor of 3.5 (50.5 s / 14.2 s).

Figure 2 displays several interesting things. First, a single rule packet (represented by the 6th bar from the top) was the only one running beyond 7 seconds. This means that the last 50% of the run time made no use of parallelism since only a single rule system task was running. In fact, this task was processing a 25-rule packet, the largest rule packet in the system.

Another point to note is that there was a substantial ramp-up time before all 51 tasks were put on the system-wide task queue. The slope of the line that could be drawn connecting the left end of the bars representing each task represents this ramp-up time. It took about three seconds before all the tasks were queued.

8. Parallelization Within Rule Packets

Figure 2 showed that the longest rule packet in the system took almost 14 seconds to run and that the entire rule system took only 14.2 seconds. This meant that no further performance improvement of any significance could be gained unless parallelism were introduced within that rule packet. The next experiment was designed to explore the process of parallelization within rule packets.

To accomplish this we wrote a parallel version of Zetalisp's COND-EVERY for use in the rule packets. Our parallel COND-EVERY calls a function which makes use of the FUTURE primitive to parallelize task execution within rule packets. COND-EVERY takes n clauses and evaluates them in a future. The parameter n may be varied to determine the optimal n for a given application. We refer to n as the *chunking factor*. This grouping does not extend beyond the borders of the individual packets. After the rules within a packet are grouped into groups of n rules each, the remainder, if any, are processed by the supervising task.

As an experiment, a chunking factor of 3 was chosen and the results were profiled. While the largest rule packet contained 25 rules, some of the rule packets in this application are rather small. The smaller rule packets may consist of only 1, 2 or 3 rules. In that case, given a chunking factor of 3, a single chunk would include the entire packet.

Figure 3 is a *zoom shot* on a portion of a profile of an experiment with the chunking factor of three. The 9 horizontal bars in the center represent the parallelization of the largest rule packet. The 9 bars represent the nine different groups that this single rule packet was divided into (8 groups of 3 rules and 1 group with 1 rule in it). This largest rule packet now runs to completion in about 5.3 seconds, a significant improvement over 14 seconds.

We then experimented with different chunking factors. Figure 4 shows the results running the rule system with a chunking factor of 5. This achieved a run time of under 7 seconds. From experiments with different chunking factors, it was determined that 5 was the optimal chunking factor for this particular application.

Figure 5 shows the effect of reversing the order of the the packets at run time. The new order lengthened the run time. We did not pursue this area of experimentation but can make some observations from a study of the task display windows. Under the given set of circumstances, the worst performance would result from an ordering of the rule packets which queued up the longest tasks last. Clearly the best performance would result when the longest tasks were the first queued. A heuristic that ordered tasks according to the anticipated length of the task would likely result in improved performance in any rule system with tasks that ran to completion.

9. MAPCAR Delay

Note that in Figure 4 there is a full 1 second delay after the first task starts before any additional task starts. (This is not the case in Figure 5 because we had already introduced the improvement we will describe in this section.) This occurs while MAPCAR is setting up the rule packets to run in FUTUREs. It appeared that MAPCAR caused this delay because of its right-to-left evaluation of arguments in SCHEME. In an attempt to eliminate this initial delay, the system code was rewritten so that a new version of MAPCAR, mapcar-new, evaluates its arguments from left to right. Figure 6 shows a test of the original MAPCAR with empty packets. The 1 second delay before the start of the second task is clearly visible. Figure 7 shows the results of running the same empty packets with the new version of MAPCAR. The delay was reduced to a fraction of a second.

The original experiment, using Data SET A, was run again using mapcar-new. The same chunking factor and packet order as in Figure 4 were used. The resulting run time was slightly over 6 seconds, as shown in Figure 8. The 1 second delay between the start of the first task and the start of the second task has been virtually eliminated. The slight delay that is still present is typical of the delays between the start of all the other tasks. It is part of the ramp up discussed in the next section.

At this point we felt we had thoroughly investigated the bottlenecks to speedup within the constraints we had established at the outset. We now ran the experiment four times to establish a final benchmark for parallel operation to compare to our benchmark for serial operation. The results are shown in Figure 9. The average run time for the tests was 6.4 seconds. This represents a performance improvement over the serial benchmark by a factor of almost 8 (50.5 s / 6.4 s = 7.9).

As for our efforts to eliminate bottlenecks, the performance improvement from the initial parallel runs to the final parallel runs was by a factor of 1.8 (14.2 s / 7.9 s).

To demonstrate that a chunking factor of 5 was, in fact, optimal for this application, we ran the experiment

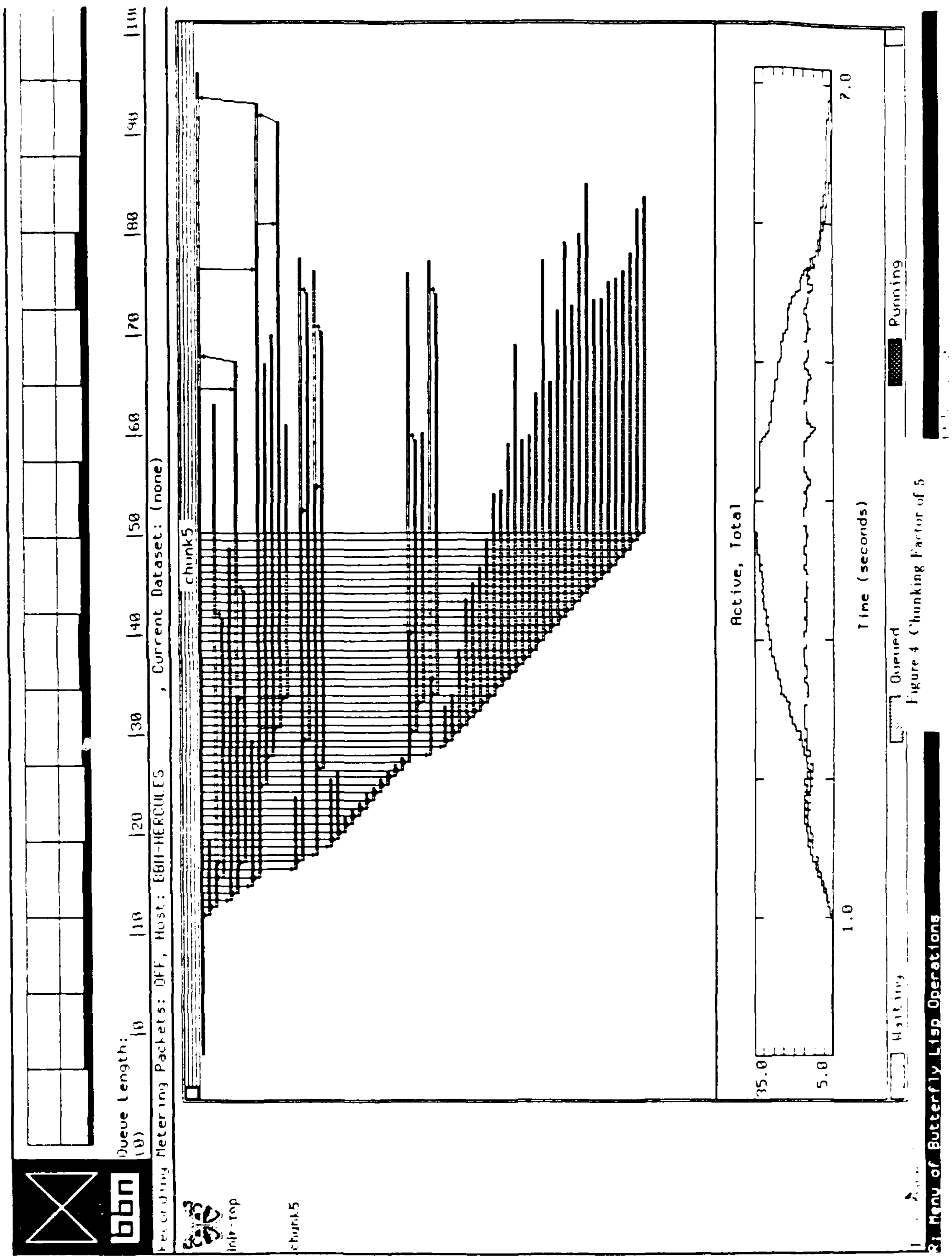


Figure 4 Chunking Factor of 5

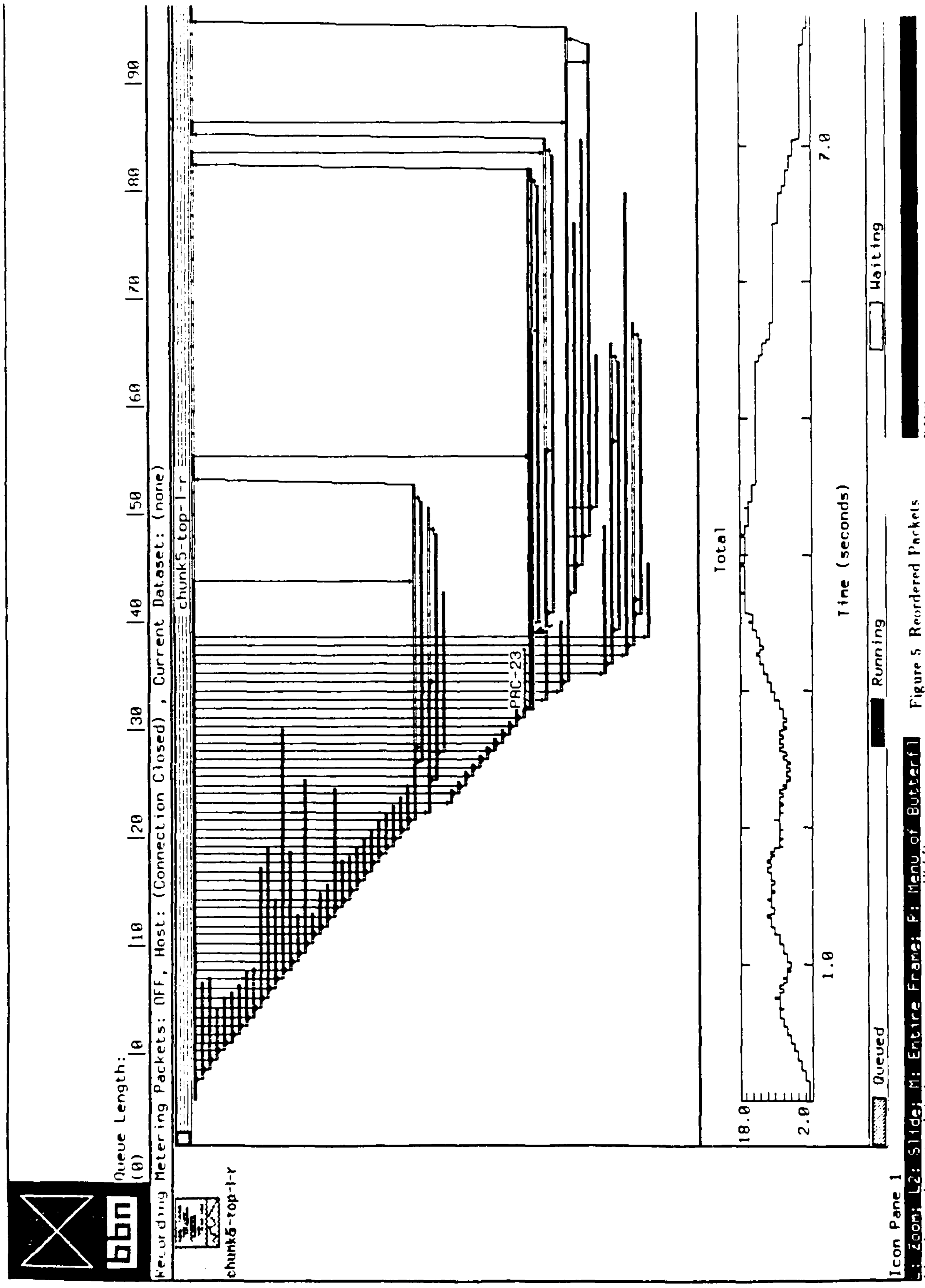
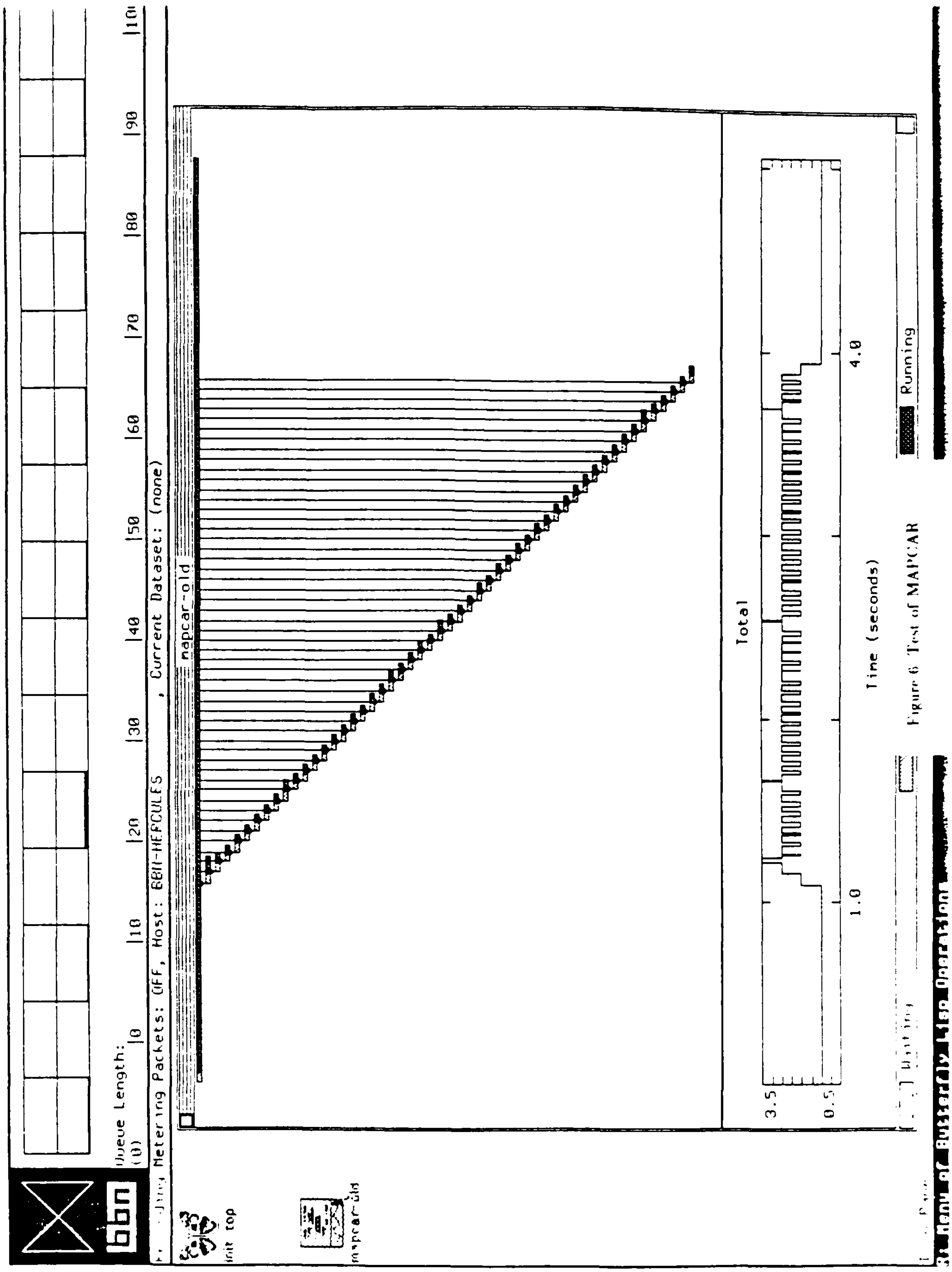
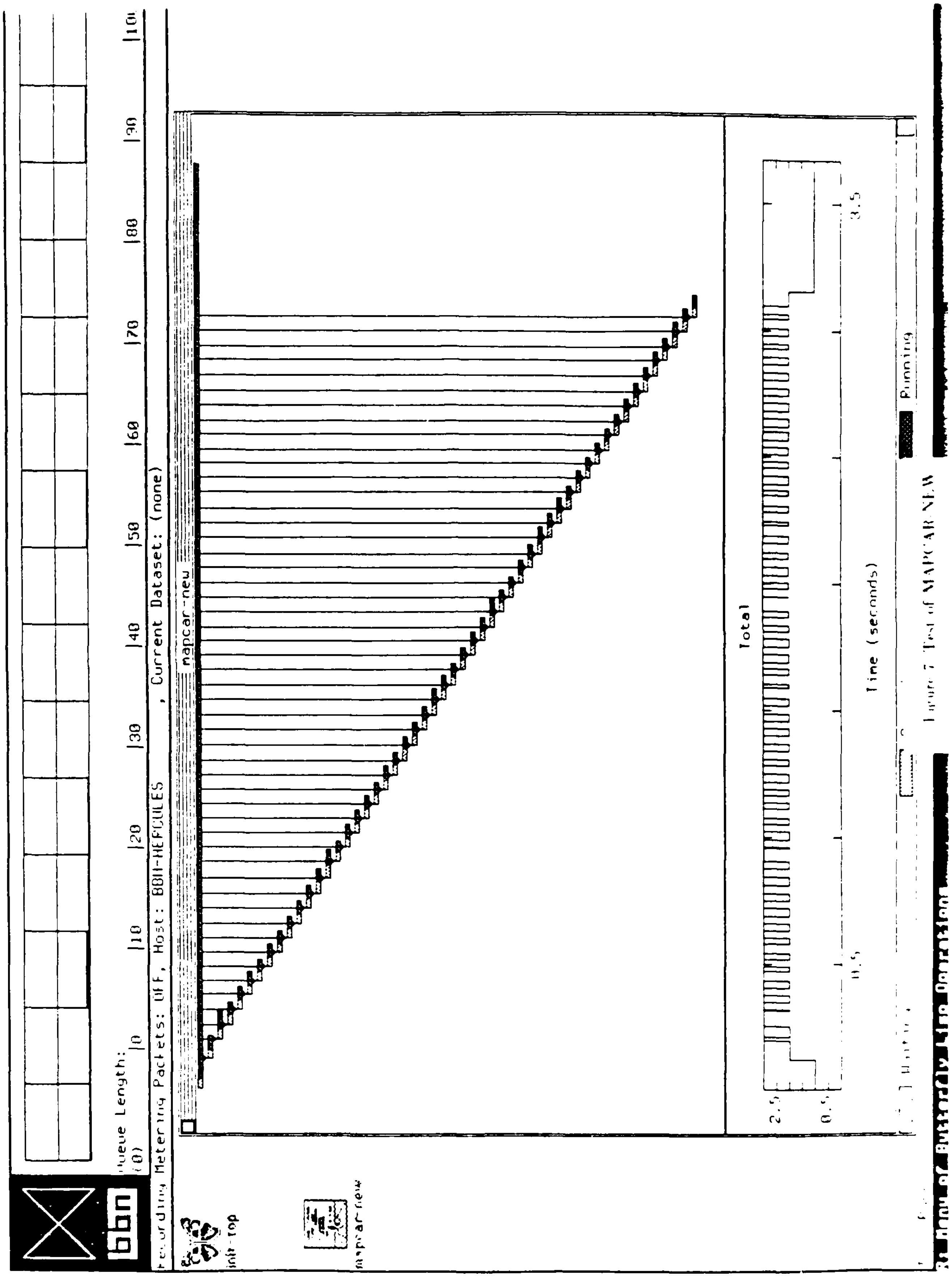


Figure 5 Reordered Packets



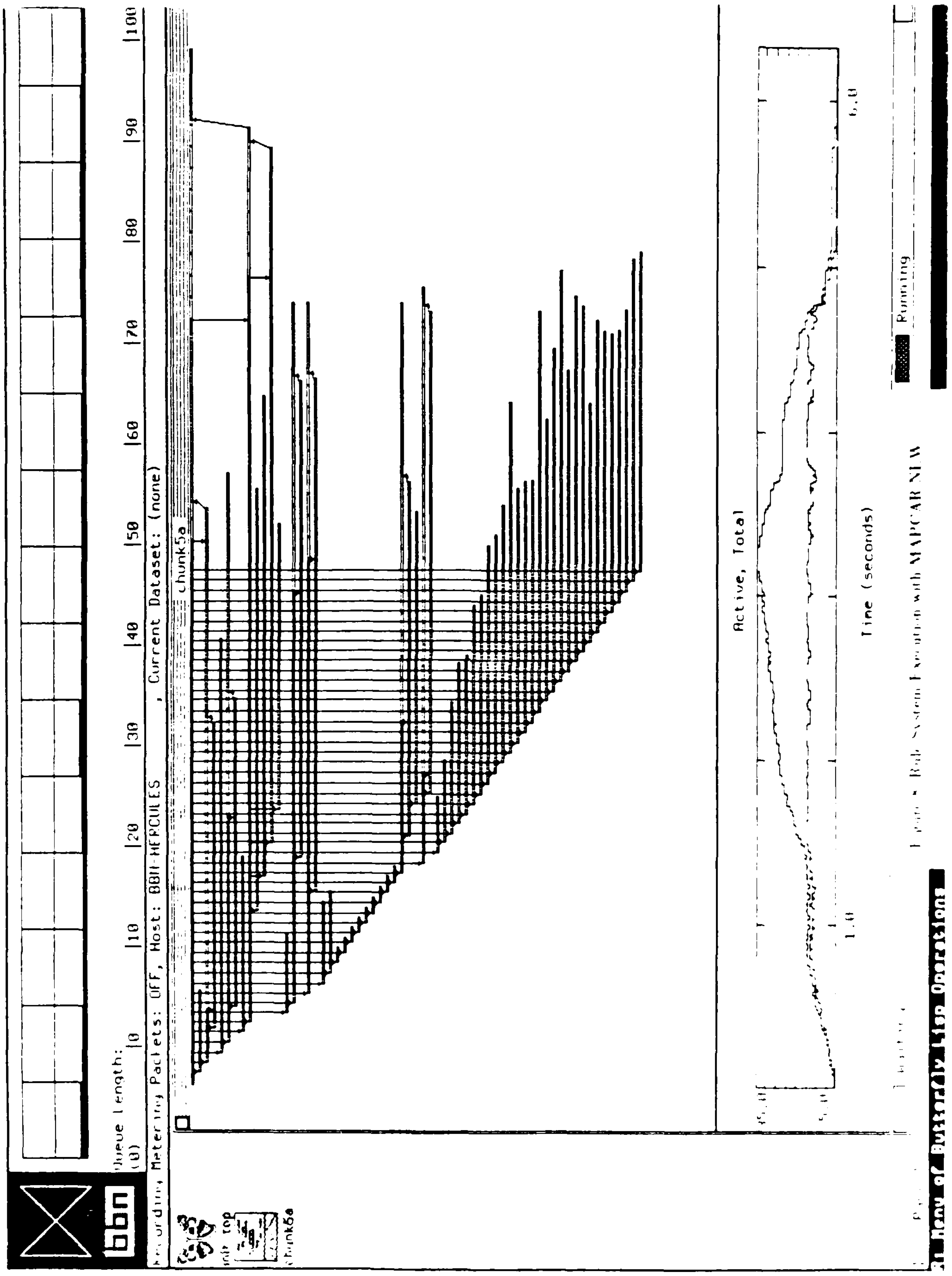
Metering Packets: OFF, Host: BBN-HERCULES , Current Dataset: (none)

Figure 6 Test of MAPCAR



Recording Metering Packets: OFF, Host: BBN-HERCULES , Current Dataset: (none)

Figure 7 Test of MAPCAR-NEW



Recording Metering Packets: OFF, Host: BBN-HERCULES , Current Dataset: (none)

Queue Length: (0)

chunk5a

Active, Total

Time (seconds)

Figure [illegible]: Rule System Execution with MAPCAR NEW

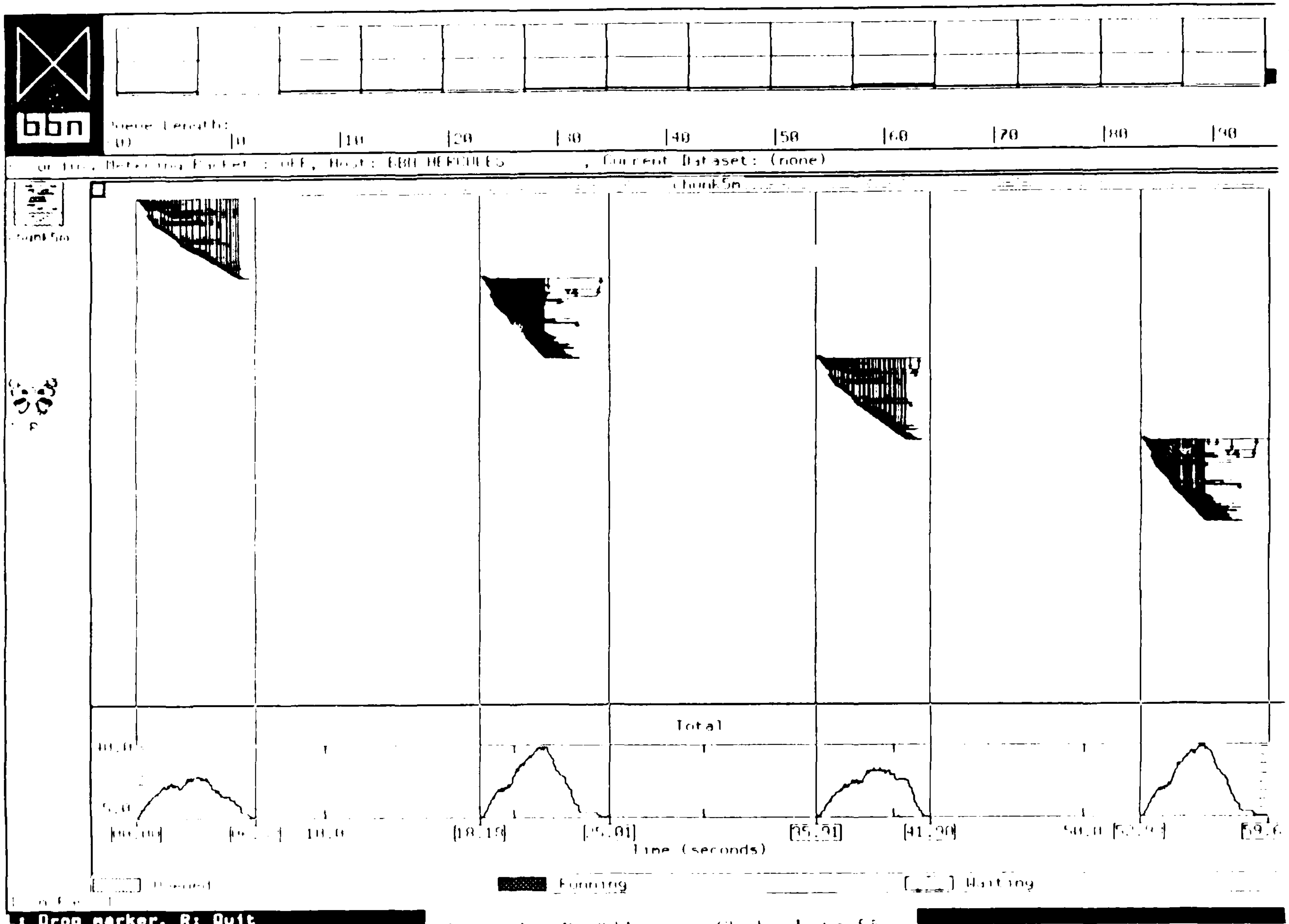


Figure [illegible]. Four Parallel Executions: Chunking Factor of 5

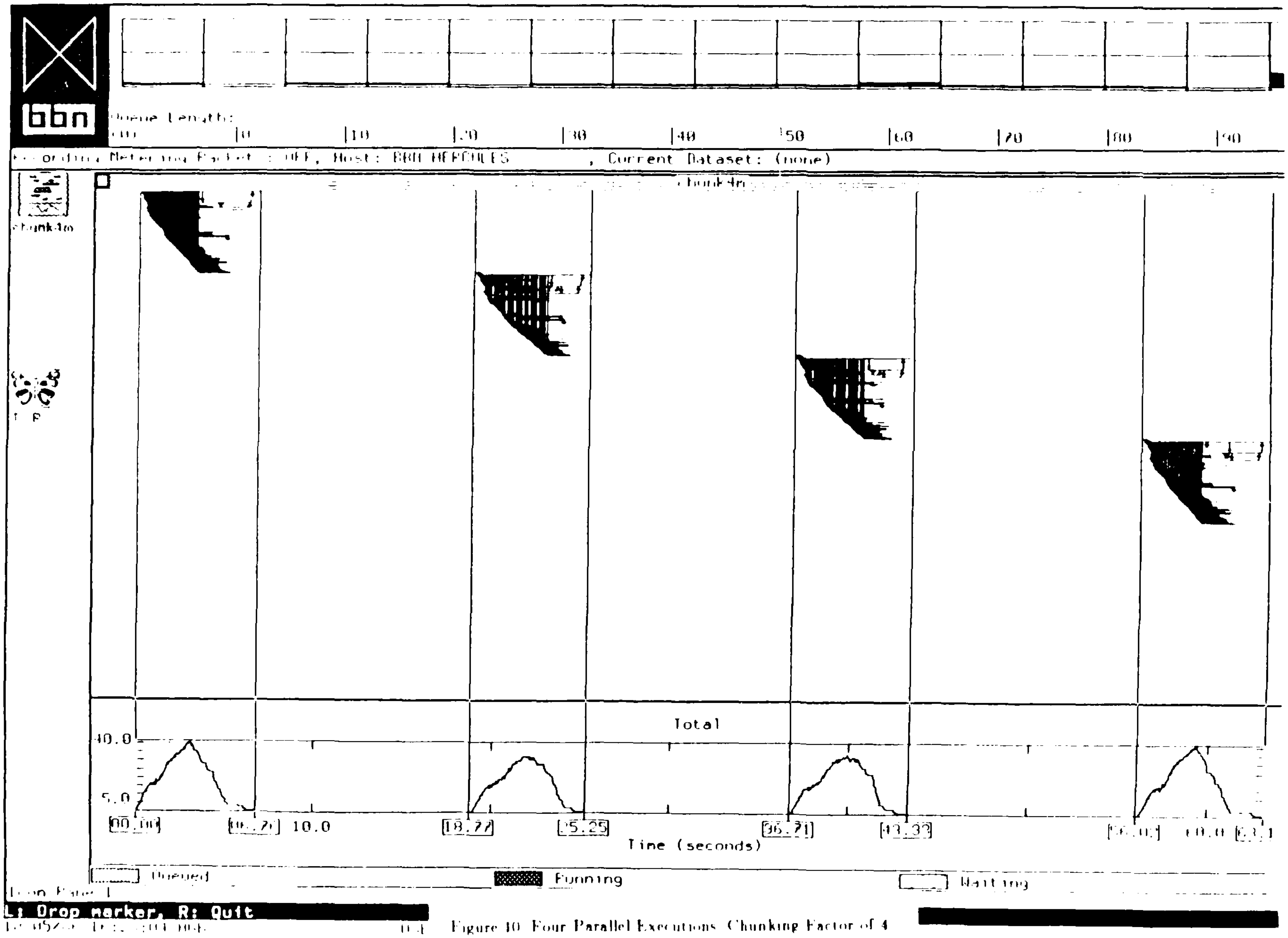
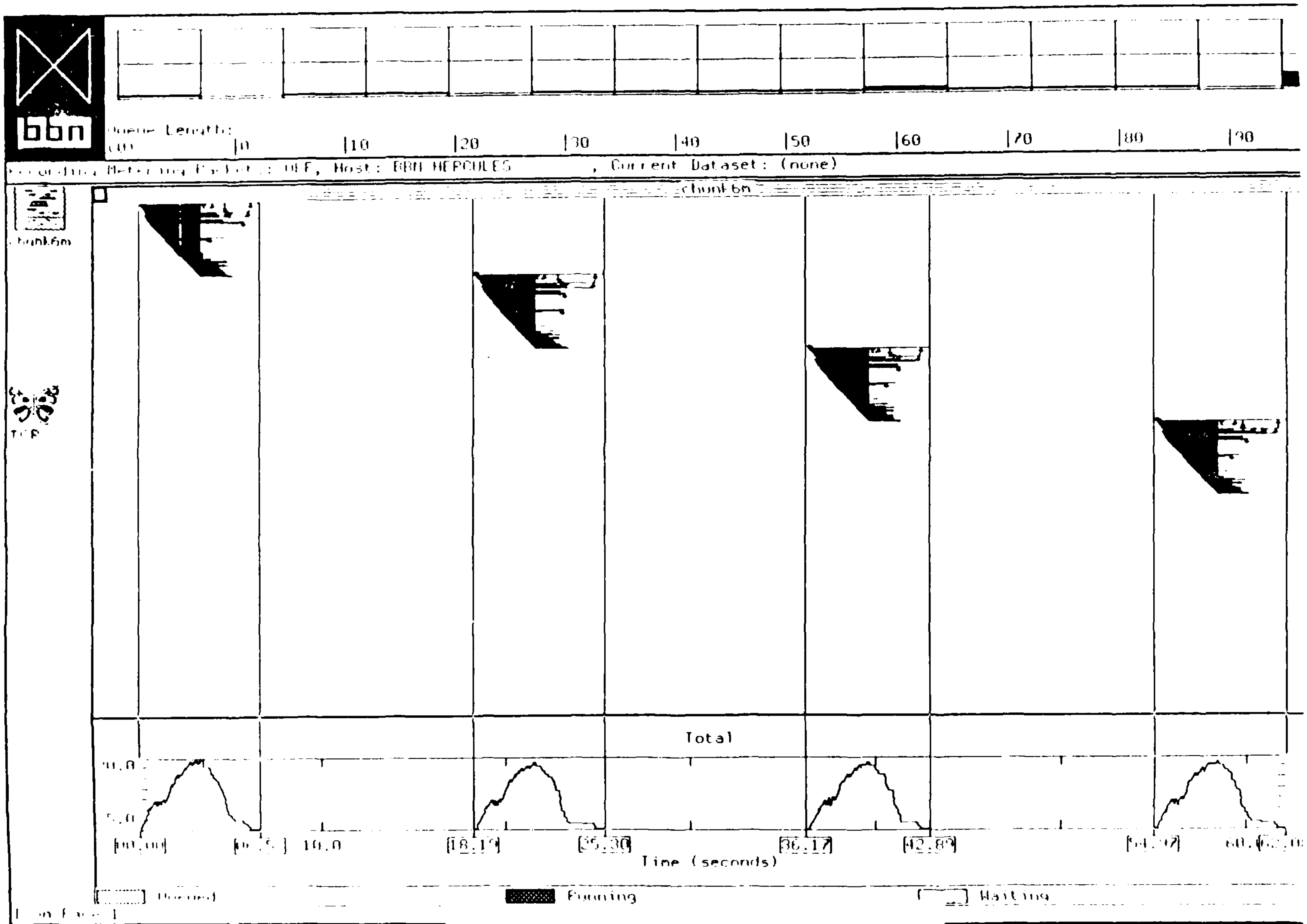


Figure 10 Four Parallel Executions Chunking Factor of 4



Recording, Metering Packets: OFF, Host: BBN HERCULES , Current Dataset: (none)

Figure 11 Four Parallel Executions Chunking Factor of 6

four times each with chunking factors of 4 and 6 to frame the results for 5. The results are shown in Figures 10 (chunking factor of 4) and 11 (chunking factor of 6). The average time for four runs with a chunking factor of 4 was 6.8 seconds. With a chunking factor of 6, the average was 7.1 seconds.

10. Ramp Up

Some additional experiments were performed in an attempt to reduce the ramp-up time. Although the ramp-up time was reduced, we were surprised to discover that this did not reduce the runtime.

The ideal shape for the profile of a set of tasks running in parallel is a square wave. This would represent a set of tasks that all began simultaneously, ran for the same length of time, and all ended at the same time. This would mean instantaneous ramp up and instantaneous ramp down. While this ideal is an unreachable goal in the real world, we experimented with making some improvements to our ramp-up time. It was apparent from the task displays that the ramp up was caused by the time spent *cdring* down a list and making and running futures for the packets (using mapcar). We reduced the processing involved in this by converting the list to an array and partitioning the array within subtasks which start up futures. The results are shown in Figure 12.

While this did improve ramp up, it did not improve the overall performance of the program. This was due to the fact that the slower ramp up resulted in a better presentation of tasks to the task queue. Figure 13 was produced by zooming in on a section of the task graph pane shown in Figure 12. The detail is now sufficient to distinguish periods when a task is queued (light, hatched area), running (dark, crosshatched area), or waiting (light, dotted area).

It is clear from Figure 13 that much of the gain that would have been realized by accelerating the process of making and running futures was lost because of the additional time that tasks spent on the task queue.

11. A Faster FUTURE

Figures 14 and 15 show an attempt to create a faster version of FUTURE. Figure 14 shows 51 empty instances of the current future. The slope on the left shows the delay involved in creating the 51 futures. Figure 15 shows the new version of FUTURE we wrote in an effort to make it faster. The left slope has been improved by one second. We did not use this faster future, because we realized that, as in the case of ramp-up, any time gained would be lost in the presentation of tasks to the task queue. We would expect that further experimentation would have led to an optimum ramp-up slope for this faster FUTURE.

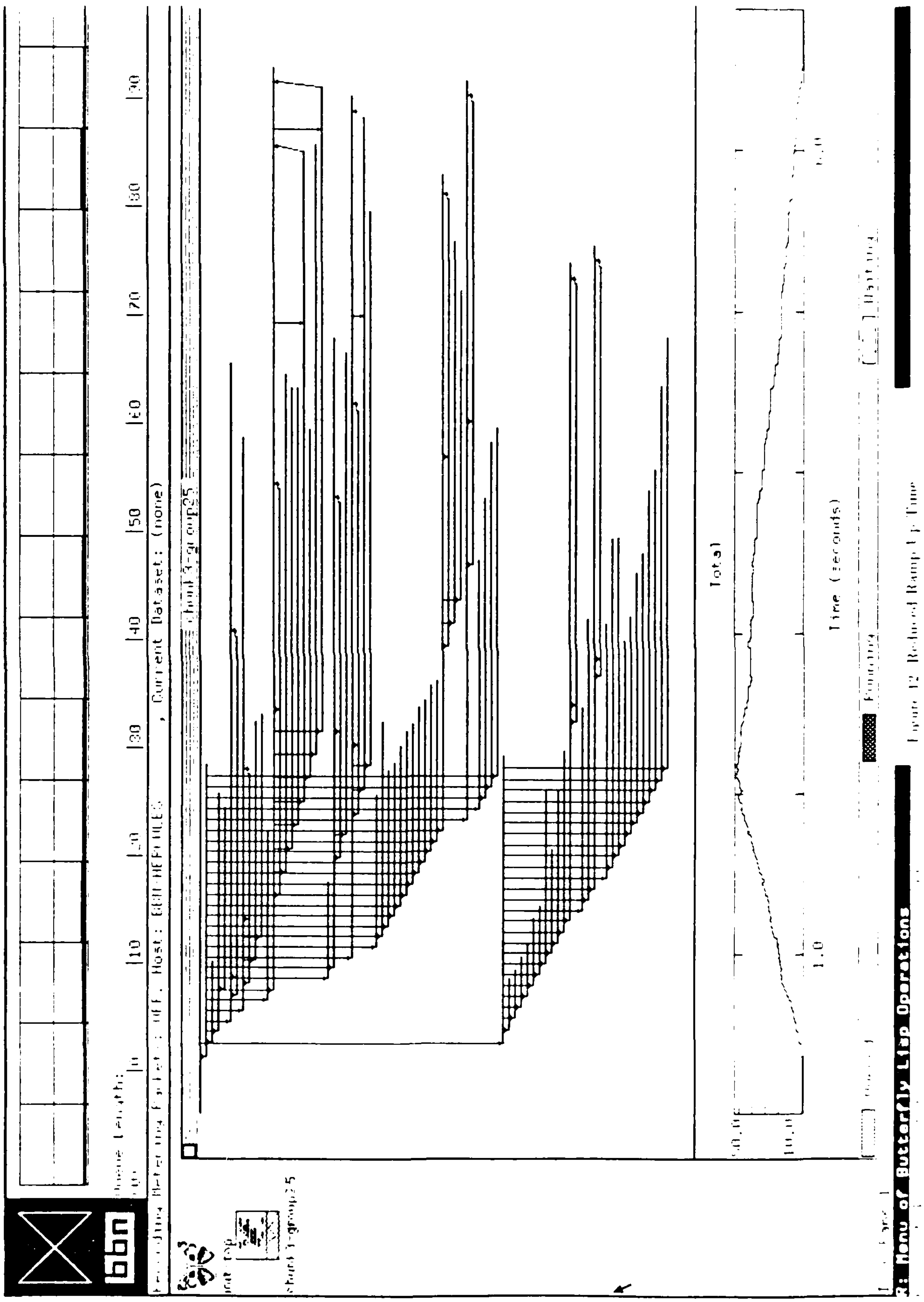
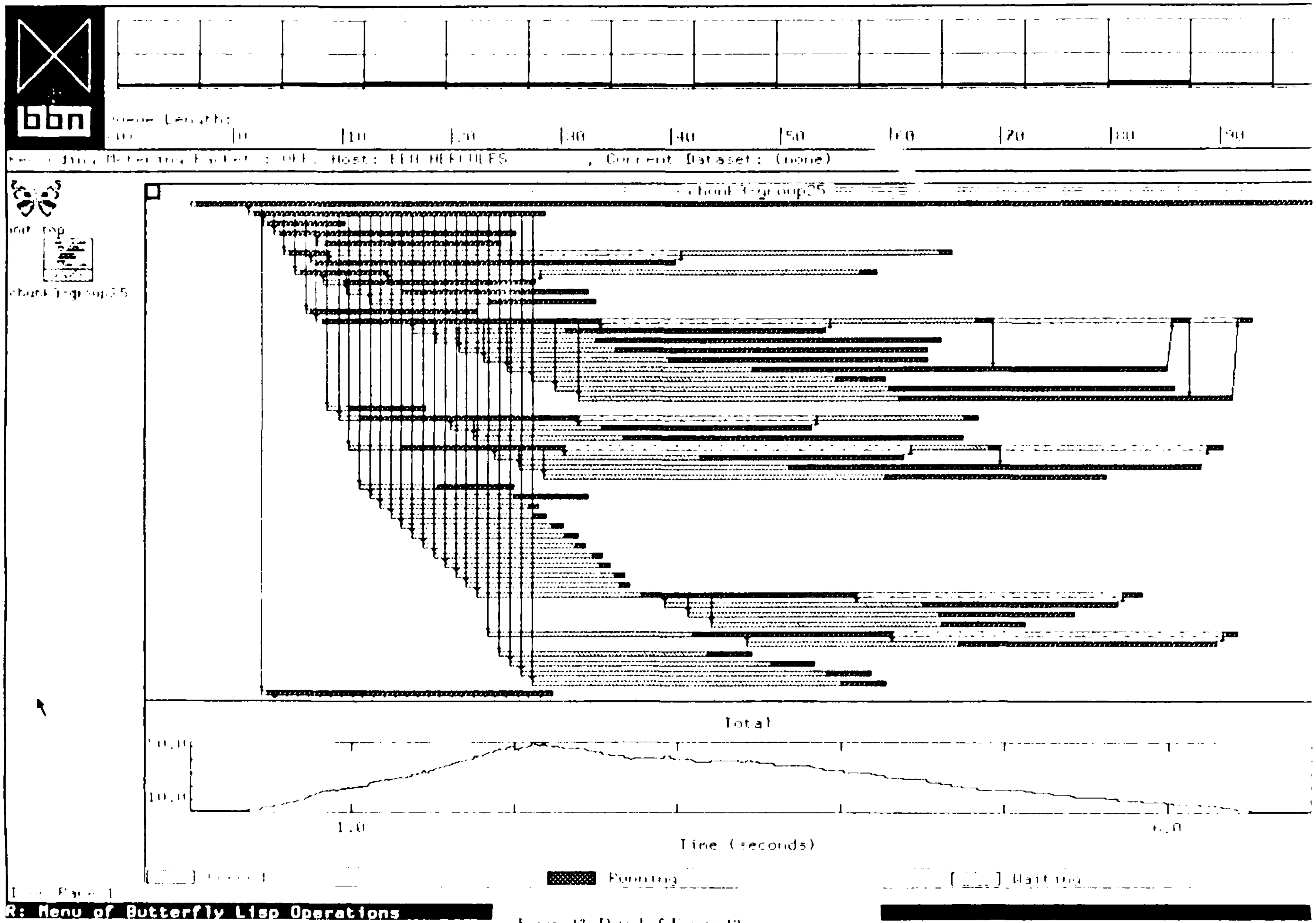

Queue Lengths
Recording Metering Packets: OFF, Host: BBN-BEETLES, Current Dataset: (none)
chunk 3-group25
|0
|10
|20
|30
|40
|50
|60
|70
|80
|90
Total
1.0
Time (seconds)
Running
Waiting
Menu of Butterfly Lisp Operations

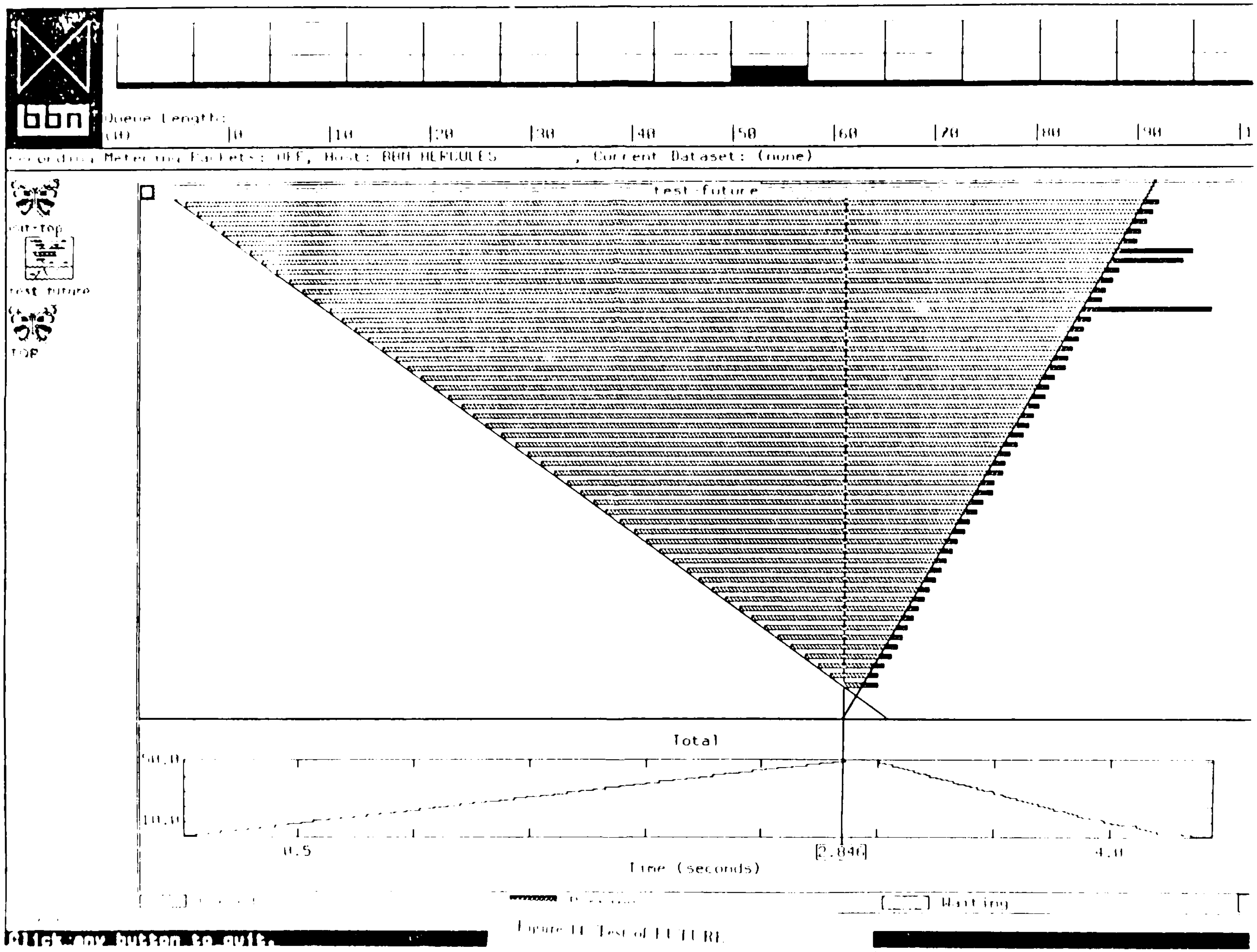
Figure 12 Reduced Ramp Up Time



Recording Metering Packet : OFF, Host: BBN HERCULES , Current Dataset: (none)

R: Menu of Butterfly Lisp Operations

Figure 13 Detail of Figure 12



Recording Metering Packets: OFF, Host: BBN HERCULES , Current Dataset: (none)

Click any button to quit.

Figure 11 Test of FUTURE

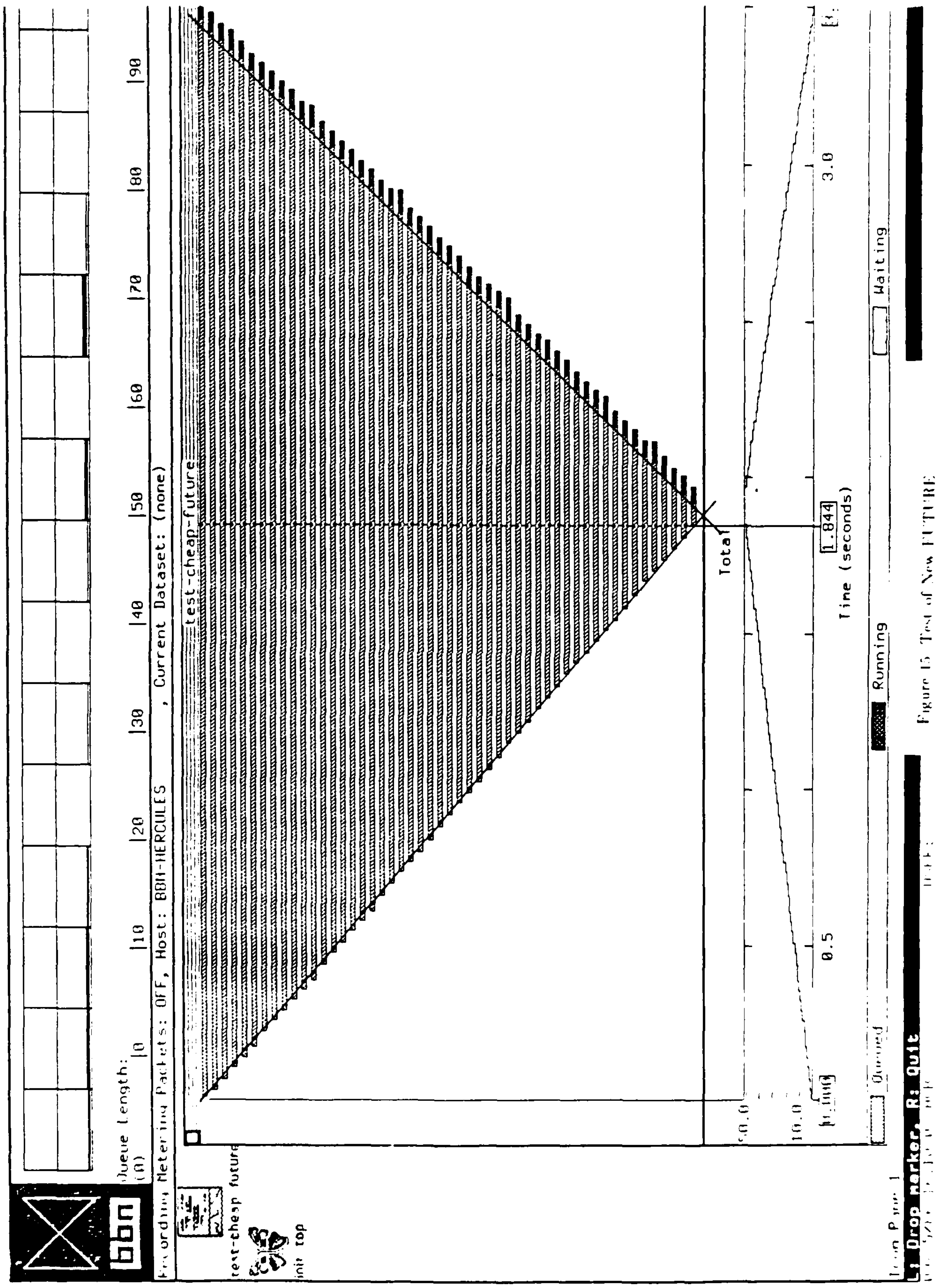


Figure 15 Test of New FUTURE

12. Data Independence of Performance Improvements

The final experiment was designed to verify that the performance improvements achieved by parallelization were not dependent on the particular data (Data Set A) used. A second data set, Data Set B, was used to determine if a comparable improvement was achieved. Whereas Data Set A fired 6 rules, Data Set B fired 15.

Figure 16 shows a profile of the program running in serial against Data Set B. Its run time was 65.4 seconds. This provided a benchmark for determining the performance improvement factor, with Data Set B as data. Figure 17 shows the optimized parallel program running against Data Set B. Its run time was approximately 7.8 seconds. This represents improved performance by a factor of about 8.3 (65.4 / 7.8).

13. Summary of Results

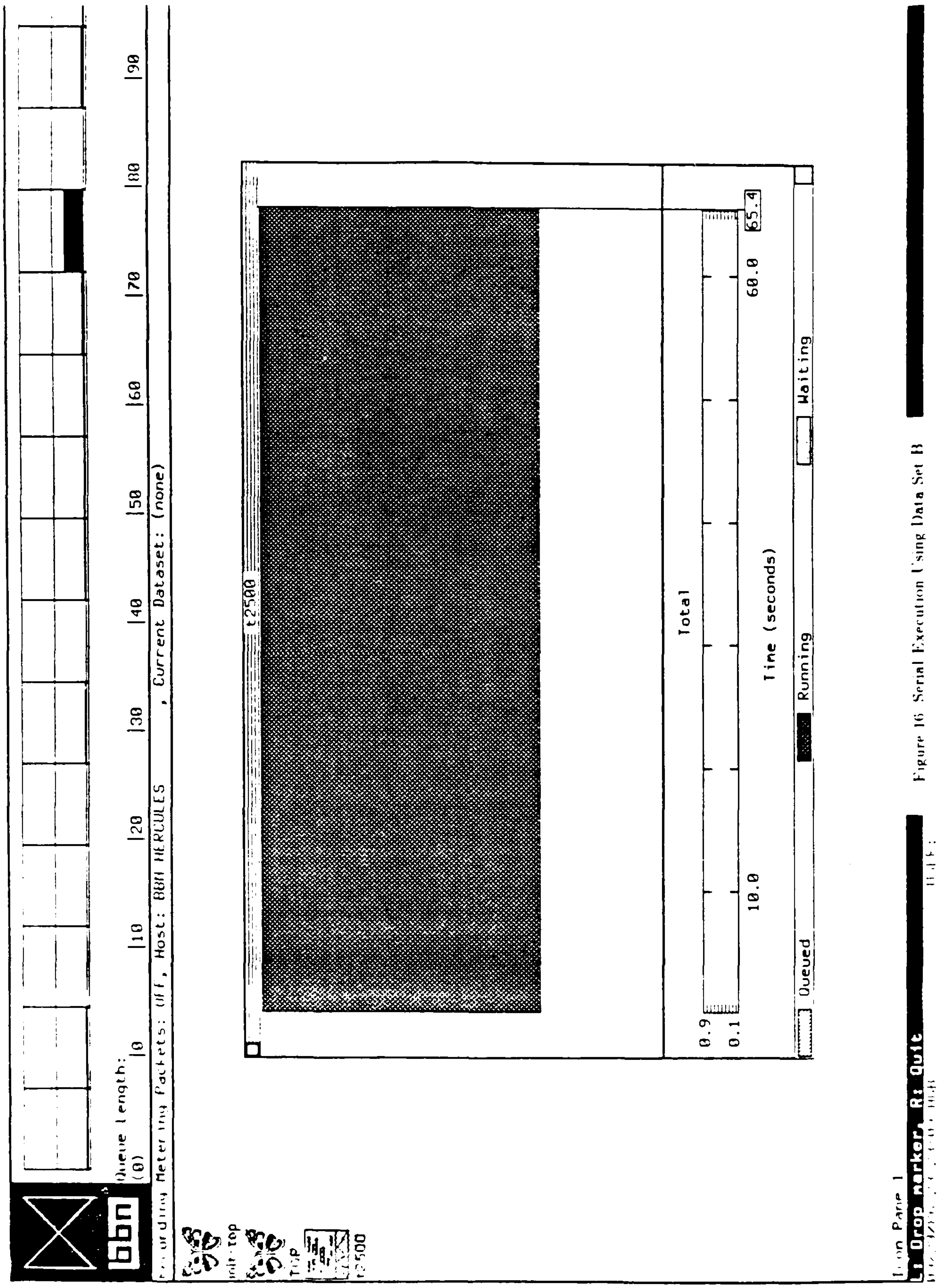
This project demonstrated that substantial performance improvements can be achieved by parallelizing a routine knowledge rule system with the FUTURE primitive and functions built with the FUTURE primitive. It also demonstrated that the task displays produced by the metering tools on Butterfly LISP provide the kind of information necessary to explore various parallelization techniques.

This project also suggest a number of areas for further research in the implementation of rule systems on parallel multiprocessors.

One route worth considering is to implement time slicing on the individual processors. Currently, once a task on the system-wide task queue is assigned to a processor, it runs to completion on that processor without interruption. If tasks are very small, or they are all approximately the same size, then this non-time slicing approach is very efficient. However, in an application where tasks vary significantly in size, a time-slicing approach would be much more efficient.

In a time-slicing approach, each processor would switch tasks at regular intervals. The processor would place the current task on its stack and process the next task until it was again interrupted after the regular interval of time and again switched tasks. The appropriate time intervals to slice the processing into, and the optimal number of tasks for each processor to juggle could be dependent on the application.

Another area for further investigation is task granularity. This system was running on a 16-processor Butterfly computer. Once the program had been reduced to a granularity sufficient to keep all 16 processors busy, there was no motivation to break the program down into a finer granularity. The desirable granularity for a program running on a parallel processor is a function of the number of processors available. If a 32- or 64-processor system were available to run this system on, the optimal granularity would probably be much finer.



Recording Metering Packets: Off, Host: BBN HERCULES , Current Dataset: (none)

Figure 16 Serial Execution Using Data Set B

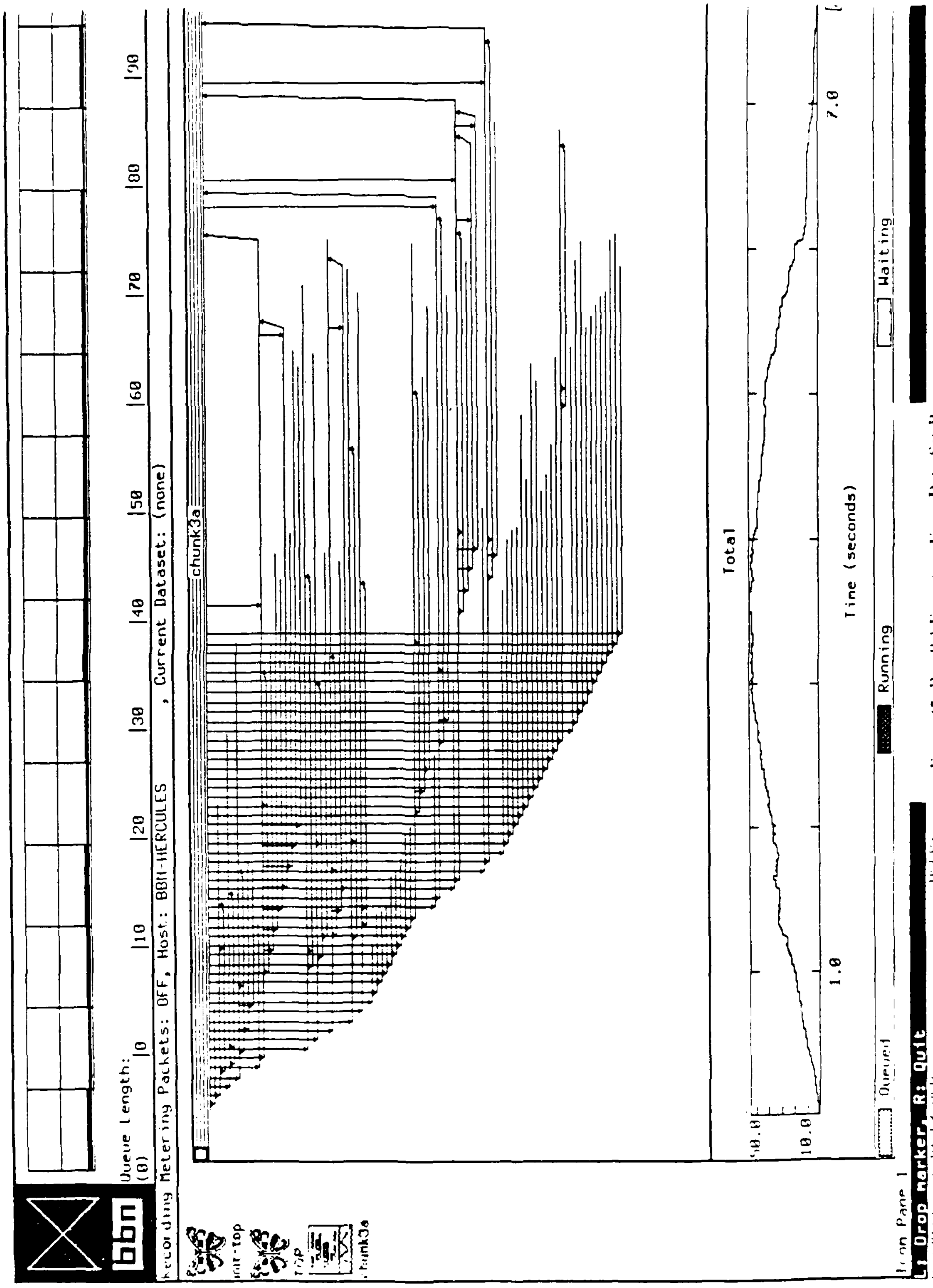


Figure 17 Parallel Execution Using Data Set B

14. Implications of this Project

Work done at CMU has raised doubts as to whether rule systems can take advantage of large scale parallelism. Our work suggests that rule systems of a certain type -- routine knowledge rule systems -- will be able to. We hasten to add that the limitations of routine knowledge rule systems are considerable and that they do not have the power of the systems analyzed in the CMU work.

We would argue that it is precisely the routine nature of the knowledge incorporated in our rule system that makes it suitable for substantial parallelization. We see no barrier to large scale parallelization of routine knowledge rule systems, provided that sufficiently large routine knowledge rule systems are developed. We agree with the conclusions of the work done at CMU, that parallelization of the actions of productions would not yield large-scale parallelism.

14.1 CMU's Work

The CMU work we are referring to was reported in a series of papers by Anoop Gupta [4], Anoop Gupta and Charles Forgy [5], and Kemal Oflazer[6]. All of this work either used production systems developed using OPS5 or assumed an OPS5-like software architecture.

Gupta and Forgy's paper, dated December 1983, reported measurements on serial production systems developed using OPS5. This work was done as part of an effort to explore the possibility of using parallelism in the execution of production systems. In the conclusion to that paper they hinted at what they felt would ultimately limit concurrency in implementations of production systems on parallel architecture: "For example, we show that actions of productions do not have global effects, but only affect a small number of productions. Furthermore, the number of productions affected is independent of, and does not increase with, the total number of productions present in the system."

Gupta's paper, dated March 1984, was theoretical and analyzed the predicted performance of three different algorithms for developing parallel production systems on the highly parallel tree-structured architecture of DADO. The DADO machine architecture was developed at Columbia University. It consists of a very large number of processing elements interconnected to form a binary tree. It was specifically designed to be suitable for the parallelization of production systems. Gupta's paper included the following conclusions:

"We argue that large-scale parallelism is not appropriate for OPS5-like productions systems (not just for DADO, but in general.) Our argument is based on the following two observations:

Measurements show that parallelism in OPS5-like production systems is limited. This is because in current production system programs, actions of a production do not have global affects, but only affect a small number of productions (approximately 35 productions are affected by each action). Furthermore, the number of productions that is affected is independent of the number of productions in the program.

Large-scale parallelism in an architecture has a strong influence on the power of the individual processsing elements. Large-scale parallelism almost always implies that each processing element is weak, that is, has narrow datapaths, has only a small amount of memory, and has a large cycle time (because most logic families with high integration density are also slow). For example, each processing element in the prototype DADO has only 8 bit wide databpaths, has only 8 kilobytes of memory, and the instruction cycle time is 2 microseconds."

Oflazer's paper was a further analysis of parallel execution of production systems. His conclusion was:

"These results indicate that contrary to most expectations, the effective parallelism in the class of production system considered, is very low compared to the number of productions in these systems. This hints that massively parallel approaches are not particularly suited to this problem. Another important observation is that the potential performance bottleneck in the production systems interpretation is not the process of selection which productions should be processed as a result of a working memory modification, but actually the process updating the states of a few productions selected. These observations point to the conclusion that in order to obtain substantial speed-ups in the interpretation of production systems faster processors and exploiting lower level overlapped processing are also necessary in addition to the exploitation of limited parallelism in the problem structure."

Oflazer also indicated that the limits on parallelism were due to the small number of productions affected by any given action: "These results point out that during execution, a given action can affect the state of only a small subset of productions, at least for the production systems analyzed. Hence the production level parallelism that can be *exploited in these production systems* (without doing useless or redundant computation i.e., performing matches whose results can be determined at compile time) is very small compared to the number of productions, therefore massively parallel approaches to exploiting production level parallelism are not particularly suited."

Our measurements suggest that Oflazer's point is a generalizable one. The two data sets we used fired only 6 rules and 15 rules, respectively. This is in a system with 221 rules. We agree that it is unlikely that large-scale parallelism can be achieved solely through parallelizing the effects of rule firings.

Oflazer's reference to matches whose results can be determined at compile time refers to the RETE algorithm which was developed to reduce the number of matches required during execution of a production system. This is a case where potential parallelism was actually reduced by a clever algorithm which made unnecessary the matching of all productions after every action.

These studies make it clear that, with the exception of possible concurrency at the lower level, parallelization of OPS5-like production systems is limited by the number of productions affected by a given action. Gupta indicates that on the average, 35 productions are affected by each action, regardless of how large the production system is. This means that parallelism on the scale of 1000 processors, or even 100 processors, or an equal number of tasks, could not be taken advantage of.

14.2 Routine Knowledge Rule Systems

Our routine knowledge rule system is less powerful than OPS5, which is a Turing machine-equivalent programming language. Rather than increasing concurrency by increasing the number of rules affected by an action, our system increases concurrency by requiring that rules be independent of each other.

Our position is that routine knowledge can be implemented in a rule system in which the variables in a rule packet that are accessed in the left hand side of the rules in that packet are disjoint from the variables modified in the right hand side of the rules in that packet. We view this characterization of routine knowledge rule systems not as an implementation-imposed limitation, but as a reflection of the nature of the routine knowledge that all human beings -- even experts -- use in most of their decision making. It is fortuitous that this independence of rules from each other makes routine knowledge rule systems candidates for large-scale parallelism. All that is needed are routine knowledge rule systems sufficiently large to take advantage of large scale parallelism. In our view, with the exception of database-type bottlenecks, there are no theoretical limitations to the amount of parallelism that can be achieved with routine knowledge rule systems.

We can imagine rule systems that are partitioned into routine knowledge sections and what we might call *nonroutine knowledge* sections. They could equal the power of an OPS5-like production system in their nonroutine knowledge sections while maximizing parallelism in their routine knowledge sections.

In closing, we'd like to speculate that the theoretical limitations to parallelism in nonroutine knowledge rule systems, limitations that were well documented by Gupta, Forgy and Oflazer, are as valid for human beings as they are for computer systems. Knowledge that has not been routinized consists of chunks of knowledge that have sequential data dependencies. Human beings, like computers, can handle routine knowledge rules in parallel because these rules are independent of each other. Nonroutine knowledge involves some sequential processing.

References

[1] Quayle, Casey and Albert Boulanger, Dawn Clarke, Michael Thome, Marc Vilain, Ken Anderson. *Butterfly Expert Systems Execution Environment: A Functional Specification*. BBN Laboratories Inc. Report No. 6225, August 1986.

[2] Shapiro, R. *FLEX - A Tool for Rule-Based Programming*. BBN Laboratories Inc., Report No. 5643, May 1984.

[3] Scott, Curtis Alan, with Don Allen, Laura Bagnall, Jim Miller and Seth Steinberg. *Butterfly LISP Reference Manual*. BBN Laboratories Inc., April 1986.

[4] Gupta, Anoop and Charles L. Forgy. *Measurements on Production Systems*. Carnegie-Mellon University, Department of Computer Science Report CMU-CS-83-167, December 1983.

[5] Gupta, Anoop. *Implementing OPS5 Production Systems on DADO*. Carnegie-Mellon University, Department of Computer Science Report CMU-CS-84-115, March 1984.

[6] Oflazer, Kemal. *Parallel Execution of Production Systems*, in *International Conference on Parallel Processing, IEEE*, August 1984.

END
DATE
FILMED
3-88
DTIC